N89-16360

A small evaluation suite for Ada compilers

Randy Wilke, Daniel Roy

1 INTRODUCTION

After completing a small Ada pilot project (OCC simulator) for the Multi Satellite Operations Control Center (MSOCC) at Goddard last year, we recommended the use of Ada to develop OCCs.

To help MSOCC transition toward Ada, we recently developed a suite of about 100 evaluation programs which can be used to assess Ada compilers, namely:

- o Compare the overall quality of the compilation system (e.g., ease of use, complexity, impact on the host computer, error message quality).
- o Compare the relative efficiencies of the compilers and the environments in which they work (e.g., how long does it take to compile and link a program?).
- o Compare the size and execution speed of generated machine code.

Another goal of the benchmark software was to provide MSOCC system developers with rough timing estimate for the purpose of predicting performance of future systems written in Ada.

2 SUITE DESCRIPTION

Two types of benchmarks were created, "static" and "dynamic". Static benchmarks are used to assess the extent to which a compiler helps (or hinders) the software development effort. Dynamic benchmarks measure the efficiency of machine code generated by Ada compilers.

The Ada evaluation suite was developed in about 4 man-months on a Digital Equipment Corporation (DEC) VAX-11/785 using the DEC Ada Compilation System (V1.0) running under the VMS operating system (V4.2). The evaluation suite source was then ported from the VAX to a Data General Corporation (DG) MV/4000 via magnetic tape. The software was rebuilt on the MV/4000 using the DG Ada Development Environment (V2.3) running under the AOS/VS (V6.3) operating system.

2.1 Static Benchmark Programs

Two general classes of static evaluation programs were generated. The first set of programs measures the time to compile various Ada constructs such as:

- o A null program to measure the minimum overhead.
- o A program instantiating INTEGER_IO.
- o A program translated from Reference 2, dealing with stride and non-stride array references.
- o The DHRYSTONE synthetic benchmark program from Reference 3.

A compilation command procedure automatically measures the compile time for every program of the benchmark suite.

The second set of static benchmark programs contain deliberately induced errors in the source code. They are used to subjectively evaluate how well compiler messages help the programmer identify some common mistakes such as:

- o Incorrect dereferencing of an object in a procedure call.
- o Confusing type and subtype declarations.
- o Common typos (missing "--" and ";", reference to a misspelled variable, etc.)
- o Forgetting to qualify items from "withed" packages. In this case, a good message should mention the right package(s).

2.2 Dynamic Benchmark Programs

The dynamic benchmark programs measure the run time overhead for the following Ada features:

- Control structures (CASE, IF-THEN-ELSE, LOOP).
- Assignment statements including ACCESS types.
- Procedure call overhead (including calling another language from Ada).
- Dynamic memory allocation.

- Sequential IO.

- Rendezvous (inter-task communication) and task activation.

- Using multi-tasking to overlap IO with CPU intensive processing.

- Array referencing (stride and non-stride).

The chosen limited set of tests concentrates on the Ada language features that are vital to MSOCC. However, the benchmark methodology and the benchmark code structure provide a good framework to easily create new benchmarks as the need arises.

An averaging technique is used to smooth the effects of random system events that can be minimized but not eliminated from the multi-programming environment. A "null" loop is timed for several iterations to compute the overhead for the loop. The ADA construct to be benchmarked is then timed inside the same loop. The null loop time is subtracted from the time of the loop containing the Ada construct, and the result is divided by the number of iterations to produce the time for one execution of the ADA construct. All timing is performed using the CALENDAR.CLOCK routine.

A command procedure automatically logs all sysgen parameters as well as the main process parameters (quotas, working set, etc.) before running all tests with a programmable number of iterations. Timing results are computed internally by every benchmark program and logged in individual files (one such file per test).

2.2.1 Parallelism Test Programs Description

The programs that test the overlapping of input, output and CPU processing with tasking warrant a more detailed discussion:

2.2.1.1 PAR_BIG

This program instantiates the SEQUENTIAL_IO package for a file of big record size (10_000 bytes per block) and reads, processes and writes several records, overlapping sequential access input, CPU intensive "processing" and sequential access output by using Ada tasking with rendezvous. The overall run time should be compared to the overall run time for SER_BIG described below.

If the compiler correctly implements the Ada tasking paradigm, the processing task should be able to run while the I/O tasks are blocked. Therefore, PAR_BIG should run faster than SER_BIG provided that the rendezvous overhead is acceptable.

2.2.1.2 SER_BIG

This program instantiates the SEQUENTIAL_IO package for a file of big record size (10_000 bytes per block) to serialize sequential access input, CPU intensive "processing" and sequential access output in a loop.

2.2.1.3 PAR_NB And SER_NB

The same principles were applied to a file of nascom blocks (600 bytes). However, because modern operating systems very efficiently buffer the data during sequential IO operations, the efficiency advantage of tasking may be small (or non existent) for this test.

2.3 Code Optimization Issues

One major concern, when doing simple dynamic benchmarks, is the compiler optimizer. Most simple benchmark programs do not do any reasonable work. One must be careful that the optimizer does not recognize this fact and optimize the construct being benchmarked completely out of the program. Even if the construct is still present, there is concern as to whether the optimization would have taken place in a "real" program to the extent that it took place in the simple benchmark (e.g., all variables used in the benchmark ending up in registers may not be realistic).

The DEC Ada Compiler has two optimization switches. One, /OPTIMIZE=TIME will automatically treat small subroutines as though the INLINE pragma had been invoked. The other, /OPTIMIZE=SIZE performs all other optimization but does not do automatic INLINE processing. The /OPTIMIZE=TIME switch does not result in automatic INLINE processing if the body of the subroutine being called is compiled separately.

We tried a method described in Reference 4 to trick the compiler into not performing automatic INLINE processing. We rejected the method because it introduced large delays that would have made timing measurements of small constructs very imprecise.

We compiled all dynamic benchmarks with and without optimization. Where significant differences resulted, the generated machine code was examined to determine if the optimizer did its job "too well". In such cases, the non-optimized version was used in test runs.

3 COMPARING DEC ACS AND DG ADE

We were guests on both of the host machines and hence, were assigned limited resources. Consequently, much effort was spent managing resources, particularly disk space. On the ADE we were frequently running at reduced priority, relative to all other system users.

This comparison between the ACS and the ADE is, perhaps, a little unfair to the ADE. The VAX-11/785, which the ACS runs on, is about twice as fast as the MV/4000 (1.2 MIPS vs 0.6 MIPS). Also, while DG's AOS/VS is far superior to many operating systems, we believe that DEC's VMS, in general, provides a significantly better software development environment. These bias must be taken into consideration.

All static and dynamic benchmarks were developed on the DEC VAX-11/785 and ported to the DG MV/4000. There were no cases where the ADE failed to compile a program that was successfully compiled under the ACS. There was one instance where the ADE generated incorrect code, and one program experienced runtime problems that were never solved. Specifically, the following problems were encountered while porting the benchmark suite:

- Due to bad code being generated for an explicit type conversion, PAR_NB had to be recoded.

- PAR_BIG never ran successfully on the MV/4000.

- File IO and parallel processing programs had to be modified on the the MV/4000 because the ADE does not handle representation clauses for type "byte" and generated code for 32 bit integer instead.

- An unhandled exception would randomly occur while using a program (written in Ada) to unpack records from files that had been transferred to the DG. The problem would go away by rerunning the program with exactly the same input file.

The following additional subjective comparisons can be made:

1. Both systems use a lot of resources. The ADE makes extravagant use of disk space and is also a CPU hog.

2. The MV/4000 text editor (SED) didn't seem as friendly as the VAX's (EDT). This may have been due to lack of DG experience on the part of the evaluators (we did not know how to use SLATE).

As a rule, setting up command files to build and run things frequently took an order of magnitude longer on the MV/4000.

3.1 Static Evaluation

3.1.1 Compilation Times

ACS and ADE compilation times for a subset of the benchmark suite are compared in Figure F.3.3-1. For the sample, the ACS performed better even if we allow for the difference in processor speeds. Differences in the time required to perform disk IO is an additional, hard to quantify factor.

The entire benchmark suite was compiled and linked in less than 40 minutes on the ACS and in about 3 hours on the ADE.

	COMPILE TIME (seconds)	
Benchmark	ACS (VAX 11-785)	ADE (MV/4000)
COMP_NULL	6	24
COMP_COMMENTS	5	32
COMP_INT_IO	11	65
COMP_TEXT_IO	7	20
ARRAY_REF	31	142
MODULO_BYTE	28	92
RV_ARRAY_100	20	94
SUB_CALL_0	18	105
PAR_BIG	69	264

Figure F.3.3-1, A Sample of Compilation Times.

3.1.2 Error Messages

Even though the ACS compile time messages were verbose at times, their relevance and clarity were judged superior to those of the ADE.

In particular, the ACS makes generally good suggestions (adding missing semicolons, guessing package name for missing qualification, etc.) whereas the ADE suggested that a derived type was intended when the problem was a confusion between type and subtype declarations. This kind of suggestion can greatly confuse the novice programmer.

3.2 Dynamic Evaluation

Overall, the DEC ACS produced more efficient code than the DG ADE. The rest of this section compares execution speeds for several classes of benchmarks.

3.2.1 Common Features

Figure F.3.3-2 shows the measured run time for the most common Ada constructs.

CONSTRUCT	ACS/ADE OVERHEAD (microsec)		
	average	low	high
Control			
3 CASES	2.6/1.5	2.6/0.8	2.6/1.9
10 CASES	2.9/1.3	2.9/1.3	2.9/1.6
IF/THEN/ELSE	4.6/1.6	4.4/1.6	4.7/1.6
FOR LOOP (optimized)	1.5/6.0	1.5/6.0	1.7/6.0
Assignments			
VARIABLE := VARIABLE	0.7/3.4	0.6/3.4	0.7/3.5
ACCESS VARIABLE := VARIABLE	3.0/5.4	3.0/5.0	3.2/5.4
VARIABLE := CONSTANT (CONST < 2**8)	0.7/2.6	0.7/2.6	0.8/2.6
VARIABLE := CONSTANT 2**8 < CONST < 2**16	1.1/2.6	1.1/2.5	1.3/2.6
VARIABLE := CONSTANT (CONST > 2**16)	1.0/2.9	0.9/2.9	1.1/2.9
Synthetic benchmark			
DHRYSTONE	1.3/4.6	1.1/4.6	1.7/4.6

Figure F.3.3-2, Common Ada construct run time overhead.

3.2.2 Procedure Call

Figure F.3.3-3 shows the run time overhead for procedure calls.

NUMBER OF PARAMETERS	PARAMETER TYPE	ACS/ADE CALL OVERHEAD (microsec/call)		
		average	low	high
0	-	13/31	13/31	13/31
1	IN	17/37	16/36	17/37
1	OUT	16/37	16/37	16/37
1	INOUT	20/40	19/40	20/40
1 (C calls C)	IN	13/NA	13/NA	14/NA
1 (Ada calls C)	IN	15/NA	15/NA	16/NA
10	IN	56/89	56/89	56/172
10	OUT	55/89	55/89	55/90
10	INOUT	86/121	86/121	86/124
10 element array	IN	14/33	14/33	14/34
10 element array	OUT	14/34	14/33	14/35
10 element array	INOUT	14/34	14/33	14/35
100 element array	IN	14/33	14/33	14/33
1000 element array	IN	14/34	14/34	14/34
10000 element array	IN	14/NA	14/NA	14/NA

Figure F.3.3-3, Procedure Call Overhead.

3.2.3 Memory Allocation

Figure F.3.3-4 shows the overhead measured for dynamic memory allocation.

NUMBER OF BUFFERS	SIZE OF BUFFERS	ACS/ADE ALLOCATION OVERHEAD (millisec/allocation)		
	(bytes)	average	low	high
100	1000	0.9/5.0	0.8/4.0	1.2/5.0
500	1000	2.9/4.6	3.6/4.6	2.8/4.6
1000	100	0.2/1.5	0.2/1.5	0.2/1.5
1000	500	6.4/4.7	6.5/4.6	6.2/4.9

Figure F.3.3-4, Dynamic Memory Allocation.

3.2.4 Sequential File IO

Figure F.3.3-5 shows the run time overhead measured for sequential IO.

RECORD SIZE	ACS/ADE IO TIMES					
(Bytes)	(milliseconds/read)			(milliseconds/write)		
	average	low	high	average	low	high
4	0.6/7	0.6/7	0.6/7	0.5/5	0.5/5	0.5/5
600	4.0/50	3.0/50	5.0/50	11/120	8.0/120	13/120
10000	120/130	100/130	140/130	340/280	300/280	400/280

Figure F.3.3-5, Sequential IO.

3.2.5 Tasking

Figure F.3.3-6 shows the run time overhead measured for a rendezvous between 2 tasks.

NUMBER OF PARAMETERS	PARAMETER TYPE	ACS/ADE RENDEZVOUS OVERHEAD (millisec/rendezvous)		
		average	low	high
0	-	1.8/11	1.8/11	1.8/12
1	IN	1.8/11	1.8/11	1.8/11
1	ACCESS	1.8/11	1.8/11	1.8/11
10	IN	1.8/12	1.8/12	1.9/12
10 element array	IN	1.8/11	1.8/11	1.8/11
100 element array	IN	2.0/12	1.9/12	2.0/12
1000 element array	IN	3.6/12	3.4/12	4.0/13
1k element array	INOUT	3.4/13	3.3/13	3.6/13

Figure F.3.3-6, Rendezvous Overhead.

Figure F.3.3-7 shows the run time overhead measured for dynamic task activation.

ACS/ADE TASK ACTIVATION OVERHEAD
(milliseconds/activation)

average	low	high
6.2/14	6.0/14	6.4/14

Figure F.3.3-7, Task Activation Overhead.

Figure F.3.3-8 shows the run time measured for reading, processing and writing a number of 600 bytes (NASCOM) and 10_000 bytes records (BIG BLOCKS) using tasking (PARALLEL) or not (SERIAL). Refer to the description of PAR_BIG and SER_BIG given previously for details.

	PROCESSING MODE	ACS/ADE TOTAL EXECUTION TIME (seconds)		
TWO CONTROLLERS:		average	low	high
NASCOM BLOCKS	SERIAL	3.7/NA	3.4/NA	3.9/NA
NASCOM BLOCKS	PARALLEL	4.1/NA	3.9/NA	4.4/NA
BIG BLOCKS	SERIAL	6.6/NA	6.4/NA	6.8/NA
BIG BLOCKS	PARALLEL	4.5/NA	4.3/NA	4.8/NA
ONE CONTROLLER:				
BIG BLOCKS	SERIAL	7.5/NA	7.3/NA	7.7/NA
BIG BLOCKS	PARALLEL	5.4/NA	5.2/NA	5.6/NA
NASCOM BLOCKS	SERIAL	NA/14	NA/14	NA/14
NASCOM BLOCKS	PARALLEL	NA/16	NA/16	NA/16

Figure F.3.3-8, Parallel Processing test.

Our results, obtained with the ACS, show that when separate controllers are used for the input and the output, parallelism is highest, allowing the PAR_BIG <u>multi-tasking</u> program to run more than 24% <u>faster</u> that its serial counterpart.

Excellent buffering by the OS however, makes the <u>serial</u> program for NASCOM blocks (SER_NB) run 10% faster that its multi-tasking counterpart.

Lack of time and numerous problems with an unfamiliar environment did not allow us to run PAR_BIG on the ADE.

3.2.6 An Interesting Math Routine

In Reference 3, it was shown that for two routines accessing an array in a stride and non-stride manner, the F77 compilers produced significantly slower code than the VAX FORTRAN and that all of the VMS Pascal compilers considered generated very inefficient code.

Our results, presented in figure F.3.3-9, show that **the VAX Ada code for this test is not as efficient as the VAX FORTRAN code (execution time for VAX FORTRAN is about half that for VAX Ada).** This result contradicts our own previous experience (see Reference 1) and the results of other groups. DEC Ada is often found to be faster than DEC FORTRAN V4.2 but we observe that DEC FORTRAN V4.3 produces significantly faster code and that the ACS optimizer can be improved. We hope that DEC Ada will benefit from the progress made for DEC FORTRAN.

ACS on VAX 11/785 CPU time (seconds)

	50 ITER-ATIONS	100 ITER-ATIONS	150 ITER-ATIONS	200 ITER-ATIONS
STRIDING:				
VAX FORTRAN (V4.3)	0.3	1.8	6.5	15.9
VAX ADA (V1.0)	0.4	3.7	13.5	38.0
NON-STRIDING:				
VAX FORTRAN (V4.3)	0.3	2.4	8.8	25.0
VAX ADA (V1.0)	0.3	2.8	10.2	26.9

Figure F.3.3-9, Array Reference Benchmark Execution Times.

4 CONCLUSION

In general, the ACS is a reasonable system to work with. The following positive comments can be made:

- The ACS operates in a logical, easy to comprehend manner. When assistance is required, documentation on operating the ACS is complete, accurate, and easy to use. On-line help is available.

- The LRM is generously supplemented with text and examples specific to the DEC implementation.

- The ACS is well integrated into the DEC software development and run-time environment. A run-time reference manual provides practical information about internal details of the DEC implementation and how Ada interfaces to VMS and other high-level languages.

- Compilation speed is rapid enough for serious software development (at least on a VAX-11/785).

- While ACS disk space requirements (per user library unit) are high, "garbage" files, necessary to track compilation units, were fewer than on ADE and were confined to the library directory, rather than cluttering the user's working directory.

- Run-time error messages were excellent. They were generally very specific about the true nature of the problem and provided the VMS standard trace back information.

The following negative comments can be made about the ACS:

- The Ada rendezvous mechanism, which will be critical to MSOCC realtime applications, incurred relatively high overhead.

- The ACS requires large amounts of disk space to maintain a user library.

- In the single direct comparison between VAX Ada and VAX FORTRAN (ARRAY_REF), our results suggest that in spite of its overall good quality, the DEC ACS code generator can be improved.

- While the information contained in compiler error messages usually identifies the offending line of code and the nature of the error, the messages themselves tend to be verbose and poorly worded. Much effort is required to extract the information from the message.

The following positive comments can be made about the ADE:

1. The ADE features a more extensive set of tools than the ACS (e.g., a pretty printer).

2. The library manager can produce very useful cross reference reports.

3. The symbolic debugger is friendly and more mature than other systems' debuggers.

4. The ADE features an impressive number of packages (e.g., BIT_OPS to alleviate the lack of representation clauses, CURRENT_EXCEPTION to help determine the origin of an exception) that would help alleviate some of the problems we mentioned.

5. Overall, the ADE generated less efficient code than the ACS but in a few cases, when the difference in CPU speed is accounted for, the ADE generated code of equal or better quality.

The following negative comments can be made about the ADE:

1. Run-time error messages were terrible. Frequently, system limits are exceeded during program elaboration. When this happens, the user is either presented with "Unhandled exception in library unit prog", or "Constraint error in unit main", and no additional information.

2. The compiler required a pragma or a compile switch to explicitly declare a procedure to be the main program. The concept is not part of the LRM and should not be necessary.

3. Compilation times were very slow, even considering the fact that the MV/4000 is only a 0.6 MIPS machine.

4. PAR_NB contained code which assigned an array to an array with an explicit type conversion (the arrays were declared as different types). The DG compiler generated bad code which caused the program to hard abort directly to the operating system with no Ada exception raised. PAR_NB was recoded to avoid the type conversion.

5. PAR_BIG never ran successfully on the MV/4000. An exception was raised the first time that a read was attempted on its input file. The reason for the exception was not apparent. SER_BIG did not have any problems reading the file. PAR_BIG worked correctly on the VAX.

6. The ADE doesn't support storage of 8-bit integers. It uses 32-bits for all integer variables, ignoring length representation clauses. In order to compare ADE IO benchmark results to ACS results, programs were modified on the MV/4000 to ensure that buffers were the same number of bytes.

7. There is no ADE compiler switch to turn off optimization. Such a switch is frequently necessary when working with symbolic debuggers and would have been useful in the benchmarking process.

8. The user's guide was rather thin and did not provide much insight into the ADE implementation of the Ada language.

9. The ADE LRM documentation did not include any ADE specific description or examples.

10. Some of the library files that ADE needs to configure compilation units must reside in the users working directory rather than in the library directory. Users have a hard enough time keeping their directories free of their own "garbage" files without also having to worry about the ADE's. The names generated for the ADE files have very long and arcane embedded number sequences, making them unwieldy to deal with on an individual basis.

11. The ADE makes extravagant use of disk space.

12. The MV/4000 text editor (SED) didn't seem to us as friendly as the VAX's (EDT). However, a colleague demonstrated a very impressive Ada frame driven editor the he built using SLATE's macro capability.

Overall, the ADE is usable for investigating the Ada language but many improvements are needed before it can be used as a production compiler.

4.1 For More Information

Two reports, available from the authors, document the suite and the results of the comparison between DEC's ACS and DG's ADE:

- An Evaluation Suite for Ada Compilers, Century Computing, Inc., Revision A, March 1986.

- A Comparison of the DEC Ada Compilation System and the DG Ada Development Environment, Century Computing, Inc., Revision A, March 1986.

The source code for the suite and the RUNOFF source file for the reports are also available from the authors on a VMS BACKUP format tape.

5 BIBLIOGRAPHY

1. Evaluation of Ada in the MSOCC Environment, Final Report, Century Computing, Inc., July 31, 1985.

2. Where are the Optimizing Compilers?, Wolfe/Macke, SIGPLAN Notices, V20, #11, November, 1985.

3. Dhrystone: A Synthetic Systems Programming Benchmark, Weicker, Communications of the ACM, Volume 27, Number 10, October, 1984.

4. Evaluating the Performance Efficiency of Ada Compilers, Bassman et al, Proceedings of the Washington Ada Symposium, ACM, 1985.

Randy Wilke is a senior member of the technical staff at Century Computing Inc. where he has been working since 1981. He received a Bachelor of Science in Computer Science from the University of Southern California in 1976.

Daniel Roy is a senior member of the technical staff at Century Computing Inc. where he has been working since 1983. He received the Diplome d'Ingenieur Electronicien (MSEE) from ENSEA in 1973 and the Diplome d'Etudes Approfondies en Informatique (MSCS) from the University of Paris VI in 1975.

Authors' current address:
Century Computing, Inc., 1100 West street, Laurel, Md., 20707.
Tel: (301) 953-3330.